Postel

Network Working Group
Request for Comments: 42

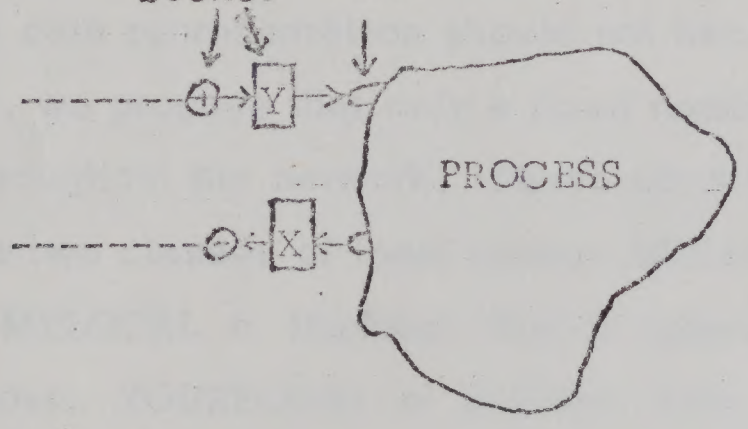
E. I. Ancona
M.I.T. Lincoln Laboratory
31 March 1970

Message Data Types

Proposal:

We propose that the first eight bits of a normal message be reserved for a message data type. Adoption of this convention does not in any way signify agreement as to the actual data types to be used. It merely establishes the convention that the first eight bits of every normal message are not available for user data.

Discussion:



It is important that conventions regarding the contents of messages be set up early so that there will not be a large proliferation of such conventions between every pair of programs running on the network.

As network usage grows, network languages may develop for specifying both the syntax and semantics of messages. However, even before such conventions are developed, a simple way of describing such a specification is by means of a message type which both sender and receiver know how to interpret.

It is important that currently running programs still run with this convention; thus, we propose that two system programs be written which initially put in and test and remove the type information from the message. Let us call these two programs X and Y, for lack of better names. In general, X and Y will perform transformations on the data, e.g., change character sets or number formats. As network usage grows, X and Y might become table driven with the table specified by the user.

Standard Types and Local Types:

We propose to distinguish between two kinds of message data types: standard and local.

Since our two transformation programs cannot be expected to perform a transformation between every possible data representation and the data representation of the machine they are running on, and also since the addition of a data representation should not necessarily involve a change to X or Y, we propose that only a fixed number of message types have meaning throughout the network. These are standard types.

There are two classes of local types: MYLOCAL and YOURLOCAL. A message type MYLOCAL n implies: this is type n of the set of types of the sending host. YOURLOCAL n implies: this is type n of the set of types of the receiving host.

Conventions:

A possible implementation of standard and local types is to define standard type 0 to be YOURLOCAL and standard type 1 to be MYLOCAL. In these cases, the second byte would be the local type number.

Local type 0 would mean user-specified, i.e., the message contents are unchanged and unchecked. Installations would define their own local type numbers and these would normally be available from the Network Information Center.

Thus initially, all messages sent to currently running programs will be type 0, n and all messages received from currently running programs will be type 1, n where n is the local type number of the character set of the installation.

Examples of Possible Standard Types:

0. YOURLOCAL
1. MYLOCAL
2. U.S. Ascii
3. EBCDIC
4. Mod 33 TTY Ascii
5. Load table driven translator table #n. If, in the future, the X and Y transformation boxes are table driven, this gives the table. The table number n is stored in the second byte of the message.
6. Use table driven translator table #n.
7. Network standard graphics message.

Examples of Local Types:

1. Local Character sets, e.g., Lincoln writer, DEC Ascii, etc.
2. Graphics local messages, e.g., TX-2 Apex display executive calls, GSAM.

NWG/RFC #42A

Postel

User View of the TSP System*

Working Paper

Enrico I. Ancona
Massachusetts Institute of Technology
Lincoln Laboratory
Lexington, Massachusetts 02173

April 1970

---

*This work was sponsored by the U.S. Advanced Research Projects Agency of the Department of Defense.

PROLOGUE

What follows is a working document on the TSP system. Much of the design is still incomplete. This document merely represents current thinking in design and should provide a basis for discussion at the 8 May 1970 meeting at M.I.T. Lincoln Laboratory.

E. I. Ancona

User View of the TSP System

I. INTRODUCTION

The purpose of the TSP (Terminal Support Processor) system is to provide a flexible I/O capability for ARPA network users. The design assumes an independent processor configuration serving as many as 20 consoles. Each console consists of a keyboard, a tablet, and a pair of storage scopes. The TSP will be connected to the network via an IMP which it shares with both the TX-2 and the 360/67 at Lincoln Laboratory.

Support services provided by the TSP system are:

(1) Input:

- Keyboard echoing and buffering
- Tablet tracking and inking
- Character recognition

(2) Output:

- Maintenance of structured display files
- Scaling and windowing on output to the display generator

Interactions between console input and output, and between TSP and other hosts on the network, are under user program control. An interpretable language called LIL (for Local Interaction Language) has been designed for the purpose of specifying the output displays as well as controlling the interactions.

LIL has primitives for manipulating display structures (for interactive graphics), as well as for message-oriented I/O. The general-purpose portion of the language is used to specify the I/O handlers and the display procedures.

Typical usage of the TSP system would be as follows: Suppose that a user wishes to write an interactive graphics subsystem to run on network host I and on TSP. In order to do so, the user writes his main computational programs to run on host I. Secondly, he writes the programs necessary for I/O in LIL. Finally, he writes also in LIL, a set of display subroutines which

he may use as primitives in the display structure he maintains on the TSP. Communication between the user's two programs is by means of message transmission primitives described in Section III.

II. DISPLAY STRUCTURE

A. Introduction

In our design of the display structure, we have been motivated by the following considerations:

(1) An interactive graphics user would like fast response to simple requests.

(2) A minimum of ARPA network communication should be required for graphics.

These considerations lead to the conclusion that the structure describing the display should be kept at the TSP and that the only graphical communication over the network should be requests to change this structure. Since certain interactive requests require no main machine communication, e.g., change the window on the display, they suffer no network delays. Furthermore, an interactive request to delete a part of the structure could be reflected quickly on the scope for the user, even though the message sent to the main machine to update the data structure there may not have been processed, or indeed arrived.

B. General Description

The display structure consists of subroutines. A subroutine consists of a list of items. Both subroutines and items have user-allocated numeric identifiers. An item contains executable code, whose format will be described later. At the simplest level, an item consists of a call, with parameters of another subroutine. Items are ordered for execution by their item identifiers.

System calls are available to add or delete both subroutines and items from the program structure. These are described later.

Thus, typically, a user would write a set of subroutines which he intends to use as primitive building blocks for his display structure. Calls to such a subroutine might cause different lines to be displayed on the scope, depending on the parameters. For example, in integrated circuit mask layout, such a routine might display a transistor and take as parameters two (x,y) pairs which describe the position and size of the transistor. In a typical application, calls on these subroutines would be added or deleted dynamically under program control.

The display structure subroutines differ only conceptually from those written to handle I/O from, say, the tablet or the network. All subroutines are called in the same manner. Thus, to repaint a picture, the user calls a particular subroutine which, perhaps via subroutine calls, causes execution of all the items necessary to generate the complete picture. Note, therefore, that there is only one locus of control for each user. It is his program structure which determines his display hierarchy. This design is possible since we are dealing with storage scopes, which do not have to be retreshed continually.

Subroutines are grouped together into program segments. In general, a user will group together logically related subroutines or perhaps he will group together subroutines which display on only a portion of his display area. Program segments have segment ids which are allocated by the system at segment load time, i.e., when the segment gets added to the TSP file system. Program segments are read only. Associated with every program segment is a data segment where the local variables for the program segment are allocated. There is a unique transformation from a program segment id to the id of its associated data segment.

Primitives exist to create and manipulate data segments which are not associated with a particular program segment.

C. System Commands for Display Structuring

The following system calls exist for manipulating data and program segments:

(1) CREATEDATASEGMENT (< segment name>, < size>, <type>)

This call dynamically creates a data segment of length < size> and of type < type> e.g., READ ONLY. It returns a segment id.

This call also enters the id and the textual name in the segment name table.

(2) CHANGEDATASEGLENGTH (< segment id>, < size>)

This call changes the length of < segment id> , to < size>.

(3) CREATEPROGRAMSEGMENT (< segment name >)

This call creates a program segment. It returns a segment id. The call also creates a data segment whose id is related to the program segment id by a unique one-to-one transformation as mentioned above. The call enters the program segment id and name in the segment name table.

(4) DELETESEGMENT (< segment id>)

This call deletes the segment, as well as the associated data segment, in the case of program segments.

(5) LOADSUBR (< segment id> , < subrid> , < n>)

This call creates, in < segment id>, a subroutine called <subrid>, containing no items, and having < n> words of local variable storage allocated to it.

(6) DELETESUBR (< segment id> , <subrid>)

This call deletes <subrid> in <segmentid> .

(7) CLEARITEMBUFFER ( )

This call initializes the item buffer, which is where items get built up.

(8) LOADCALL (< segment id> , <subrid> , < p1 >, < p2> ,...)
This call adds a subroutine call to the item buffer, with the given parameters. The parameters either may be constants, addresses, or values. These are described in Section IV. Note that the values of the parameters are those at the time the LOADCALL is executed, not at the time < subrid> is executed. In order to get the values of parameters at the time < subrid> is executed, the parameters must be addresses.

(9) LOADBUFFER (< addr > , < length>)
This call adds the block of < length> words, starting at < addr> , to the item buffer.

(10) PUTITEM (< segment id > , < subrid> , < item id >)
This call puts the contents of the item buffer in < subrid > as < itemid> . but does not clear the item butter.

(11) DELETEITEM (< segment id > , < subrid>, < item id >)
This call deletes < item id> from the subroutine.

(12) REPORT ITEM (< segment id > , < subrid > , < item id> , <addr> , < n>)
This call reports the contents of an item into a buffer of < n> words starting at <addr> .

(13) SEGID (< segment name >)
Returns the segment id.

(14) SEGNAME (< segment id> , <addr> , < n >)
Returns the segment name into a buffer of <n> words starting at <addr> .

III. THE I/O SYSTEM

A. Introduction

The TSP user's I/O capability includes two storage scopes, a keyboard, a tablet and the ARPA network.

In our design we felt it was important that the type of data transmitted or received should depend as little as possible on the particular physical device employed. Instead, we felt that a canonical representation of the data from a device should be developed. Messages to or from a device would thus be of a particular type. See RFC #42 (1) for more on message data types. The I/O system calls in LIL are based on the concept of streams of messages. A stream is a source or sink of messages. A message is one or more 8-bit bytes. What constitutes a message depends upon its type.

Certain devices may be capable of outputting or inputting several types of messages, e.g., a tablet device might produce messages consisting of characters recognized by a character recognizer, or messages of single coordinates or lists of coordinates ("ink points"). Thus, when attaching a device to a stream, the type of message desired must be specified.

Note that several input devices may be attached to the same stream and several output streams to an output device. Thus, messages of different types may be on the same stream. It is only by placing messages on the same stream that a user preserves the time ordering of message transmission.

The defined input devices on the TSP are:

(1) Keyboard

(2) Tablet

(3) Network

(4) Data File

---

(1) Network Working Group, Request for Comments #42.

The defined output devices are:

(1) Scopes

(2) Network

(3) Data File

B. Message Types

(1) Ascii: Two subtypes are defined

a) string: A message of type Ascii string consists of a sequence of characters. The number of characters is given in the first byte of the message.

b) char: A message of type Ascii char consists of a single character.

(2) Data: A message of type data consists of an arbitrary number of bytes, whether for input or output. The only restriction on the message length is the physical device the system is connected to, e.g., ~ 8000 bits for the network, 2000 words for a segment, etc. The first two free bytes, after the data type number, of a data message contains the length of the message, in 8-bit bytes.

(3) Tablet: Several subtypes are defined.

a) Coord: A message of type tablet coord consists of 4 bytes: the first two are an X coordinate, and the second two a Y coordinate.

b) Inkbuff: A message of type tablet inkbuff consists of two bytes giving the length of the message, followed by tablet coordinates describing the ink points.

c) Comparator: A message of type tablet comparator consists of two bytes giving the subroutine id, two bytes giving the item id, and 4 bytes giving the XY position.

There may be more.....

C. System Commands for I/O

The system calls which are available are:

(1) CREATEINPUT (< stream id>)

This creates an input stream with the given id, i.e., it allocates a buffer area.

(2) CREATEOUTPUT (< stream id>)

This creates an output stream.

(3) ATTACH (<stream id> , <device id> , < type>)

This attaches <device id> , to < stream id> where <type> is the type of message expected.

(4) DETACH (< stream id> , <device id> , < type >)

This detaches the device from the stream.

(5) CLOSE (< stream id>)

This detaches all devices from the stream. Buffered messages will be lost and further messages not accepted.

(6) READ (< stream id> , < buffer addr>)

Read next message from < streamid> into < buffer addr> .
If <streamid> is empty, a special message to that effect will always be in the stream.

(7) WRITE (< streamid> , < buffer addr>)

Write next message into < streamid> from < buffer addr>

(8) WAIT

Wait for a message in any stream, then proceed.

There may be more.....

IV. GENERAL PURPOSE LANGUAGE

This section contains a description of the general purpose portion of LIL in which the user codes his display subroutines and his I/O handlers.

A. TSP Virtual Machine

The virtual machine the TSP user perceives, and in which LIL runs, consists of:

a) A segmented address space. Each segment may be up to 2K words long. The user may define up to 2^{16} segments, in practice, however, he will have a small fraction of this total defined.

b) 64 general registers. These registers transcend the segment structure, i.e., they are common to all segments. Arithmetic may be performed on all the registers.

c) 8 base registers. These registers also transcend the segment structure. They contain bit patterns which will always be interpreted as segment identifiers.

B. Addresses and Data

Data in LIL consists of 16-bit words. There is no byte addressing. Two types of addresses are defined: register addresses for the 64 general registers, and segment addresses which consist of a base register address and an offset. Unless otherwise specified, either type of address may be used in any operation.

C. Pseudo-Ops

(1) Syntax: SEGMENT < name>

Semantics: This should be the first statement. It defines the name of the segment. A < name> is any sequence of letters and numbers, starting with a letter.

(2) Syntax: REGISTER { < name > = < constant >
< ; < name > = < constant > > . }

Semantics: Use < name > as the name of general register < constant > . This is a data type declaration.

(3) Syntax: SUBR < id > < n > < par > < , < par > > .

Semantics: Subroutine called < id > starts here and uses the given parameters as dummy arguments. An < id > is any 16-bit quantity. < n > is the length of the data area associated with this subroutine.

(4) Syntax: ITEM < id >

Semantics: The item called < id > starts here. The item is terminated by the next occurrence of the ITEM statement.

(5) Syntax: VEC {< name > = < value >}

Semantics: < name > is a vector of length < value > in this segment, i.e., < name > is a pointer of the 0th word of the vector.

D. Operations

The LIL instruction set is divided into three classes of operations:

(a) Arithmetic and logical operations

(b) Control operations

(c) Miscellaneous operations

1. Arithmetic Operations

N2 and N3 will always be considered addresses whose contents are required, unless square brackets [ ] are used which indicate an immediate operand.

a) Syntax: N1 ← N2 < op> N3

where < op> may be

+	add
-	subtract
$\bar{*}$	multiply upper
$\underline{*}$	multiply lower
/	integer quotient
REM	integer remainder
$\wedge$	logical AND
$\vee$	logical OR
$\oplus$	logical EXCLUSIVE OR

Semantics: (i) In each of the above, N2 may be shifted left or right by a constant, or by the contents of a general register.

For example,

$$A \leftarrow B \not\downarrow 3 + C$$

means shift B left by 3 bits

For example,

$$\text{REGISTER } \{R = 3\};\ A \leftarrow B \not\downarrow R + C$$

means shift B right by the contents of general register 3.

(ii) A * (star) next to a name implies indirect

N1, N2 and N3 may all be indirect.

For example,

$$A^* \leftarrow B^* \not\downarrow 3 + C$$

means take the contents of the address given by B, shift it left by 3, add C, and store at the address given by the contents of A. Note that the indirect occurs before the shift.

(iii) Every arithmetic statement may have ',S' ',Z' or both appended, implying that the sign or zero conditions, or both of the result is saved in the sign status or zero status bits, respectively.

For example,

A ← B + C, Z

means add the contents of A to the contents of B, and store in A. If the result is zero, store 1 in the zero status bit, otherwise, store 0.

b) Syntax: N1 ← N2

Semantics: Assign N2 to N1. Shifting N2, indirection and saving the condition bits is permitted.

2. Control Operations

a) Syntax: JUMP < s > , < z > < itemid>

JUMPREL < s> , < z> < n instr > where < instr> is a self relative address, < s> may be S, $\bar{S}$ or blank and <z> may be Z, $\bar{Z}$ or blank.

Semantics: Jump if the S and Z bits match the given bits.

e.g., 1. JUMP, $\bar{Z}$ L means jump to item L if the Z bit is 0, and the S bit is anything.

2. JUMPREL, -3 jump unconditional up 3 operations.

b) Syntax: SWITCHVIA <addr > < L1> , < L2 > ,...

Semantics: Jump to the ith label in the list, where i is the value of < addr>. A label is either a self-relative address or an item id.

c) Syntax: DO < loopvar> , < incr> , < endvalue>, < label>

Semantics: Compute < loopvar> + < incr> , store in < loopvar> , compare to < endvalue> , and jump to < label> if < loopvar> ≤ < endvalue>.

d) Syntax: CALL < subrid> < segid> < p1> ...

Semantics: call subroutine < id> with parameters < p1> , < p2> ... The return address is stored in locations 0 and 1 of the data area of the called subroutine.
Location 0 contains the calling segment id, location 1 contains the offset.
The values of the parameters are then placed from location 2 on.

e) Syntax: RETURN

Semantics: Return from subroutine call using the return address found in locations 0 and 1 of the data area of the subroutine.

f) Syntax: FINISH

Semantics: Return to "system".

3. Miscellaneous Operations

This is whatever is left.

a) Syntax: LOADBR < n> , < addr>

Semantics: Load base register < n> (< n> = 1,...,7) with the contents of < addr>.

b) Syntax: STOREBR < n> , < addr>

Semantics: Store the contents of base register < n> in < addr>.

c) Syntax: COMMAND < name > < p1 > < p2 > ...

Semantics: What follows is a system command, e.g., I/O or structuring. The name of the system command is < name > , and its parameters are < p1 > < p2 > ...

Error handling is not yet specified.